XHAMI - Extended HDFS and MapReduce Interface for Image Processing Applications

Raghavendra Kune[1], Pramod Kumar Konugurthi[1], Arun Agarwal[2],Raghavendra Rao Chillarige[2], and Rajkumar Buyya[3]

{raghav.es, pramodkumar.konugurthi, aruncs.2011}@gmail.com,vijaya_crr@yahoo.co.in,rbuyya@unimelb.edu.au

[1]Advanced Data Processing Research Institute, Department of Space, India

[2]School of Computer and Information Sciences, University of Hyderabad,India

[3]CLOUDS Lab, Department of Computing and Information Systems, University of Melbourne, Australia

Abstract– Hadoop Distributed File System (HDFS) and MapReduce model have become de facto standard for large scale data organization and analysis. Existing model of data organization and processing in Hadoop using HDFS and MapReduce are ideally tailored for search and data parallel applications, for which there is no data dependency with neighboring/adjacent data. Many scientific applications such as image mining, data mining, knowledge data mining, satellite image processing etc., are dependent on adjacent data for processing and analysis. In this paper, we discuss the requirements of the overlapped data organization and propose XHAMI as a two phase extensions to HDFS and MapReduce programming model to address such requirements. We present the APIs and discuss their implementation specific to Image Processing (IP) domain in detail, followed by sample case studies of image processing functions along with the results. XHAMI though has little overheads in data storage and input/output operations, but greatly improves the system performance and simplifies the application development process. The proposed system works without any changes for the existing MapReduce models with zero overheads, and can be used for many domain specific applications where there is a requirement of overlapped data.

Keywords: *Cloud Computing, Big Data, Hadoop, MapReduce, Extended MapReduce, XHAMI, Image Processing, Data intensive Scientific computing, Remote Sensing.*

1. Introduction

The amount of textual and multimedia data has grown considerably in recent years due to the growth of social networking, healthcare applications, surveillance systems, earth observation sensors etc. This huge volume of data in the world has created a new field in data processing called as Big Data [1], which refers to an emerging data science paradigm of multi-dimensional information mining for scientific discovery and business analytics over large scale scalable infrastructure. Big Data handles massive amounts of data collected over time, which is an otherwise difficult task to analyze and handle using common database management tools [2]. Big Data can yield extremely useful information; however, demands new challenges both in data organization and processing the data effectively [3].

Hadoop [4] is an open source framework for storing, processing, and analysis of large amounts of distributed semi structured/unstructured data [5]. The origin of this framework comes from internet search companies like Yahoo and Google, who needed new processing tools and models for web page indexing and searching. This framework is designed for data parallel processing at Petabyte and Exabyte scales distributed on the commodity computing nodes. Hadoop cluster is a highly scalable architecture, that spawns both compute and data storage nodes horizontally for preserving and processing large scale data to achieve high reliability and high throughput. Therefore, Hadoop framework and its core sub components i.e. HDFS [6][7] and MapReduce [8][9][10] are gaining popularity in addressing several large scale applications of data intensive computing in several domain specific areas like social networking, business intelligence, and scientific analytics, etc. for analyzing large scale, rapidly growing, variety structures of data.

The advantages of HDFS and MapReduce in Hadoop eco system are – horizontal scalability, low cost setup with commodity hardware, ability to process semi-structured/ unstructured data, and simplicity in programming. However, HDFS and MapReduce, though offer tremendous potential for gaining maximum performance, but due to its certain inherent limiting features, does not confine to be used for all areas. Below we describe one such domain specific applications in remote sensing image processing.

- ***Remote sensing image applications***

Earth observation satellite sensors provide high-resolution satellite imagery having image scene sizes from several megabytes to gigabytes. High resolution satellite imagery for example Quick Bird, IKONOS, Worldview, IRS Cartosat etc. [11] are used in various applications of analysis and information extraction like oil/gas mining, engineering construction like 3D urban/terrain mapping, GIS developments, defense and security, environmental monitoring, media and entertainment, agricultural and natural resource exploration etc. Due to increase in the numbers of satellites and technology advancements in the remote sensing, both the data sizes and their volumes are increasing on a daily basis. Hence, organization and

analysis of such data for intrinsic information is a major challenge.

Ma et al. [12] have discussed challenges and opportunities in Remote Sensing (RS) Big Data computing, focused on RS data intensive problems, analysis of RS Big Data, and several techniques for processing RS Big Data. Two dimensional structured representation of images, and majority of the functions in image processing being highly parallelizable, the HDFS way of organizing the data as blocks and usage of MapReduce functions for processing each block as independent map function, makes Hadoop a suitable platform for large scale high volume image processing applications.

An image is a two-dimensional function f(x,y), where x and y are spatial (plane) coordinates, and the amplitude of f at any pair of coordinates (x,y) is called intensity or gray level of the image at that point [13]. Image data mining is a technology that aims in finding useful information and knowledge from large scale image data [14]. This involves use of several image processing techniques such as enhancement, classification, segmentation, object detection etc. which use many combinations of linear/morphological spatial filters [13].

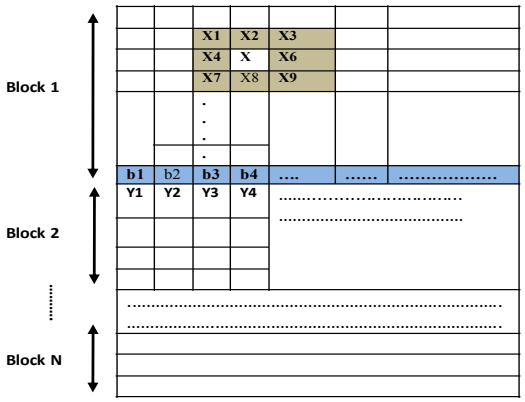


Figure 1. Image representation with segmented blocks

Many of the linear/morphological spatial filters demand use of adjacent pixels for processing the current pixel. For example, as shown in Figure 1, a smoothening operation performs weighted average of a 3X3 kernel window. The output of pixel *X* depends on the values of *X1,X2,X3,X4, X6,X7,X8, and X9*. Therefore due to the dependency, these types of operations cannot be performed on the edge pixels. Hadoop and many of the implementations discussed in Section 2, split the data based on a fixed size, which results in partitioning of data as shown in Figure 1. Each of the blocks is written to different data nodes. Therefore the boundary pixels of entire line *b1, b2, b3,..* in each block cannot be processed, as the adjacent pixels are not available at the respective data nodes. Similarly for the pixels marked as *y1,y2,y3,y4*,… also IP operations cannot be performed straight away. To process these boundary pixels i.e., the start line and end line in each block a customized map function to read additional pixels from a different data node is essential, otherwise the output would be incorrect. This additional read operations for each block increase the overhead significantly.

Section 2 describes related work in image processing with HDFS and MapReduce over Hadoop framework. Section 3 describes proposed two phase extended system XHAMI and usage of APIs. Section 4 describes experimental results, and Section 5 presents the conclusions and future work.

2. Related Work

Image processing and computer vision algorithms can be applied as multiple independent tasks on large scale data sets simultaneously in parallel on a distributed system to achieve higher throughputs. Hadoop [4]is an open source framework for addressing large scale data analytics using HDFS and MapReduce programming models. In addition to Hadoop, there are several other frameworks like Twister [15] for iterative computing of streaming text analytics, and Phoenix [16] used for map and reduce functions for distributed data intensive Message Passing Interface (MPI) kind of applications.

Kennedy et al. [17] demonstrated the use of MapReduce for labeling 19.6 million images using nearest neighbor method. Shi et al. [18] presented use of MapReduce for Content Based Image Retrieval (CBIR), and discussed the results obtained by using around 400,000 images approximately. Yang et al. [19] presented a system MIFAS for fast and efficient access to medical images using Hadoop and Cloud computing. Kocalkulak et al. [20] proposed a Hadoop based system for pattern image processing of intercontinental missiles for finding the bullet patterns. Almeer et al. [21] designed and implemented a system for remote sensing image processing with the help of Hadoop and Cloud computing systems for small scale images. Demir [22] et al. discussed the usage of Hadoop for small size face detection images. All these systems describe the bulk processing of small size images in batch mode over HDFS, where each map function processes the complete image.

White et al. [23] discussed the overheads that can be caused due to small size files, which are considerably smaller than the block size in HDFS. A similar approach is presented by Sweeney et al. [24] and presented Hadoop Image Processing Interface (HIPI) as an extension of MapReduce APIs for image processing applications. HIPI operates on the smaller image files, which are bundled

into a large block called Hadoop Image Bundle (HIB). In HIPI each image is applied to only one map function, which has limitation in dividing the data into smaller file sets. All these said methods discussed aggregation of smaller images and mapping each image within the bundle as a whole to one single map function.

Srirama et al. [25] discussed the processing small/regular images of total 48675 by aggregating them into large data set, and processed them on Hadoop using MapReduce as sequential files, similar to the one addressed by HIPI. Also, presented feasibility study as a proof-of-concept test for a single large image as blocks and overlapping pixels for non-iterative algorithms image processing. However, no design, or solution, or methodology has been suggested to either to Hadoop or MapReduce for either Image Processing applications or for any other domain, so that the methodology works for existing as well as new models under consideration.

This paper addresses the issues related to processing large remote sensing images which run into several Megabytes to Gigabytes, addressing several issues related to data organization over HDFS, and processing them by MapReduce using extended HDFS and MapReduce called as XHAMI library. The proposed extensions are applied for image processing applications, but the same can be extended to other domains also where such similar data dependency exists.

3. XHAMI- Extended HDFS and MapReduce

In this section we describe XHAMI - the extended software package of Hadoop for large scale image processing/mining applications. First we present XHAMI APIs for reading and writing (I/O), followed by MapReduce for distributed processing. We discuss two sample case studies i.e. histogram and image smoothening operations. Histogram computes the frequency of pixel intensity values in the image, and smoothening operation uses spatial filters like sobel, laplacian etc. [13].

3.1 XHAMI – HDFS I/O extensions

Figure 2 depicts the sequence of steps in reading/writing the images using XHAMI software library over Hadoop framework. Initially, client uses XHAMI I/O functions (step 1) for reading or writing the data. The client request is translated into create () or open () by XHAMI, and sent to DistributedFileSystem (step 2). Distributed File System instance calls the namenode to determine the data block locations (step 3). For each block, the namenode returns the addresses of the datanodes for writing or reading the data. DistributedFileSystem returns FSDataInput/Output Stream, which in turn will be used by XHAMI to read/write the data to/from the datanodes. XHAMI checks file format, if the format is in image type (step 4), then metadata information such as file name, total scans, total pixels, total numbers of bands in the image, and the number of bytes per pixel are stored in HBASE [27], this simplifies header information reading as and when required through HBASE queries, otherwise reading the header block by block is tedious and time consuming process.

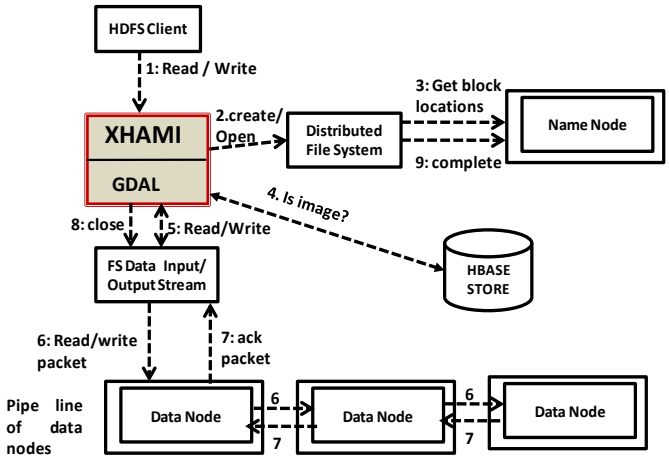


Figure 2. XHAMI for read/write operations

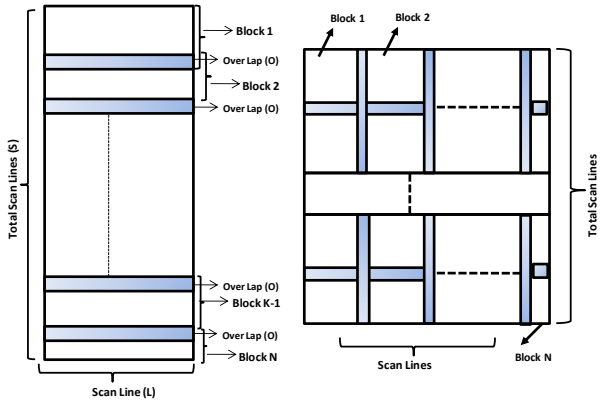
Later on XHAMI calls FSDataInput/Output Stream either to read/write the data to/from the respective data nodes (step 5). Steps 6 and 7 are based on standard HDFS data reading/writing in the pipelining way. Each block is written with the header information corresponding to the blocks i.e. blockid, start scan, end scan, overlap scan lines in the block, scan length, and size of the block. Finally, after the read/write operation the request is made for closing the file (step 8), and the status (step 9) is forwarded to the namenode.

The major XHAMI APIs for I/O related operations are described in Table 1. XHAMI reads/writes the image blocks into the same format of that original file, using Geographical Data Abstraction Layer (GDAL) library [26] during I/O and MapReduce operations.

Table 1. Description of functions in XHAMI I/O API

Functions provided
*int xhmrWriteImage(*String *file,* int *overlap)* Description: content of *file* to be written into HDFS with the specified numbers of *overlap* scan lines. This call is used for writing the files of type images. Return status: if success returns 1 else 0.
int xhmrWriteFile(String file) Description: *file* contents (which are of not image types) are written into HDFS. Return status: if success returns 1 else 0.
String[] xhmrReadFile(String file)

Description: used for writing *file of* type non image to HDFS. Return status : the contents of the file in string format
byte[] xhmrReadImage(String file) Description: reading the contents of the *file* from HDFS. Return status : the contents of the file in binary format
int xhmrReadGetTotalScans(String file) Description: returns total scan lines in the image with name *file.* Return status: the contents of the file in binary format.
int xhmrReadGetTotalPixels(String file) Description: returns the total number of pixels of the image *file.* Return status: the contents of the file in binary format.
byte[] xhmrReadGetRoi(String file, int startscan int start pixel,int blockwidth, int blockheight) Description: reads the region of interest of the image *file* starting at the *startpixel,* with a block of size *blockwidth* and *blockheight,* and returns the bytes that are read. Return status: the contents of the file in binary format.
byte[] xhmrReadGetBlockData(String file, int blocknumber) Description: returns *bytes* at the *blocknumbe* of the *file.* Return status: the contents of the block data in binary format.
String[] xhmrReadGetBlockHeader(String file, int blocknumber) Description: Returns the header in string format of the *file,* at the *blocknumber.* Return status: the contents of the header in the file at the corresponding block number.



(a) Unidirectional split **(b)Bidirectional split**

Figure 3. Block construction methods

The image is organized as blocks in HDFS with overlap among the subsequent blocks. The blocks are constructed in two ways i.e. **(i) unidirectional**: partitioning across the scan line direction as shown in Figure 3.a, and **(ii) bidirectional**: partitioning both horizontal and vertical directions as shown in Figure 3.b. while construction, it is essential to ensure that, no split take place within the pixel byte boundaries. The methods are described below.

i) Unidirectional split: blocks are constructed by segmenting the data in across scan line (horizontal) direction. Each block is written with the additional lines at the end of the block.

ii)Bi-directional split: splitting the file into blocks in both horizontal and vertical directions. The split results in the blocks, for which, the first and last blocks have overlap with their adjacent two blocks, and all the remaining blocks have overlap with their adjacent four blocks. This type of segmentation results in large storage overhead which is approximately double the size of the unidirectional segment construction. This type of organization is preferred while images have larger scan line lengths.

In the current version of XHAMI package data organization is addressed for unidirectional segmented blocks, however, it can be extended for bi-directional split. The segmentation procedure is described below.

Scan lines for each block *Sb* computed as

$$Sb = \left\lceil H/(L * P) \right\rceil$$

H = HDFS Default block length in Mbytes.
L = length of scan line i.e. total pixels in the scanline.
P = pixel length in bytes.
S = total number of scan lines.

Total number of blocks *T,* having overlap of α number of scan lines is

$$T = \left\lceil S/Sb \right\rceil$$

If $T^* \alpha > Sb$ *then* $T = T+1$.

The start and end scan lines $B_{i,s}$ and $B_{i,e}$ in each block is given below; N representing total scans in the image.

$$B_{i,s} = \begin{cases} 1, & i = 1 \\ B_{i-1,e-\alpha+1}, & 1 < i < T \\ B_{N-1,e-\alpha+1} & i = T \end{cases}$$

$$B_{i,e} = \begin{cases} B_{i,s} + S_b - 1 & 1 \le i < T \\ S_b\, i = T \end{cases}$$

Block length is computed as below.

$$R_i = (B_{i,e} - B_{i,s} + 1) * L * P\,,\ 1 \le i \le T$$

The blocks are constructed with metadata information in the header, such as blockid, start scan, end scan, overlap scan lines in the block, scan length, block length. Though, metadata adds some additional storage overhead, but, simplifies the processing activity during Map phase, for obtaining the total number of pixels, number of bands,

bytes per pixel etc, and also helps to organize the blocks in the order during the combine/merge phase using blockid.

3.2 XHAMI – MapReduce Extended Functions

In this section we describe the extensions for Map and Reduce functions for image processing applications. Based on the image processing operation either map function alone, or both map and reduce functions are implemented. For example, edge detection operation does not require the reducer, as the resultant output of the map function is directly written to the disk. Each map function reads the block numbers and metadata of the corresponding blocks. The sample job configuration, and map function are shown Table 2, and Table 3. XHAMI offers three different APIs as illustrated in Table 4.

Read operations can be implemented in two ways in HDFS, one way is to implement own split function, ensuring the split does not happen across the boundaries, and other one is to use FIXED LENGTH RECORD of FixedLengthInputFormat class. As, the block sizes are fixed, currently we have used the fixed length record format. The description of the APIs is given below.

Table 2.Sample job configuration for MapReduce

```
1. JobConfconf = new JobConf(ImageMapReduce.class);
2. conf.setWorkingDirectory(new
Path("hdfs://namenode/user/hduser"));
3. conf.addResource(newPath("/home/hduser/hadoop-
2.7.0/etc/hadoop/core-site.xml"));
4. conf.addResource(newPath("/home/hduser/hadoop-
2.7.0/etc/hadoop/hdfs-site.xml"));
5. conf.setInt(FixedLengthInputFormat.FIXED_RECORD_LEN
GTH, blocklength);
6. conf.setInputFormat(FixedLengthInputFormat.class);
```

Table 3. Sample Map function

```
public void map(LongWritable key, BytesWritable value,
OutputCollector<IntWritable,IBytesWritable> output,
Reporter reporter) throws IOException{
//code for reading the data
byte [] b = value.getBytes(); //buffer for processing
// remaining operations follows on byte b
}
```

Table 4.XHAMI processing APIs for Map/Reduce

(1) void xhmrHistogram(String inputfilename, String outputfile)
(2) void xhmrSobel(String filename, String outputfile)
(3) void xhmrLaplacian(String filename,Stringoutputfile)

(1) void xhmrHistogram(String filename, String outputfile)

Histogram operation computes frequency count of the pixel in the image. The histogram is computed as follows, first, the block and length of the block is read, and each blockis mapped to one map function. Sample code for histogram of map and reduce function is described in Table 5 and Table 6 respectively.

Table 5.Histogram map function

```
public void map(LongWritable key, BytesWritable value,
OutputCollector<IntWritable, Text> output,
Reporter reporter) throws IOException {
    byte[] data = value.getBytes();
    byte pixelValue=0;          //skip overlap scan lines
    for(inti=0;i<data.length-
    (overLapScanLines*scanLineLength);i++){
    pixelValue= data[i];
    output.collect(new      IntWritable(pixelValue),      new
    Text(""+1));
    }
}
```

Table 6.Histogram reduce function

```
public void reduce(IntWritable key, Iterator<IntWritable>
values, OutputCollector<IntWritable, Text> output,
Reporter reporter) throws IOException {
int sum=0;
while (values.hasNext()){
      sum+=Integer.parseInt(""+values.next());
}
byte b = (byte)key.get();
int v = (int)b;
key  = new IntWritable(new Integer(v));
output.collect(key,new Text(""+sum));
}
```

(2) void xhmrSobel(String filename, String outputfile)

Edges characterize boundaries in images are areas with strong intensity contrasts- a jump in intensity from one pixel to the next. There are many ways to perform edge detection. However, the majority of different methods may be grouped into two categories, gradient, and Laplacian. The gradient method detects the edges by looking for the maximum and minimum in the first derivative of the image. The Laplacian method searches for zero crossings in the second derivative of the image to find the edges. An edge has the one-dimensional shape of a ramp and calculating the derivative of the image can highlight its location. In the map function, for edge detection, the combiner and reduce functions are not performed, as there is no need of aggregation of the individual map functions. The description for the map function for sobel operator is given in Table 7.

Table 7. Sample map function of Sobel gradient operator

```
public void map(LongWritable key, BytesWritable value,
OutputCollector<IntWritable, BytesWritable> output,
Reporter reporter) throws IOException {
 //read the meta data of the block and skip the block
byte [] data1 = value.getBytes();
//declare ouputdata buffer
byte [] outputdata  = new byte[data.length];
InputStreamin= new
ByteArrayInputStream(value.getBytes());
FileOutputStreamfos = new FileOutputStream(new
File(fileName));
//apply the kernel on data buffer and write it to outputdata.
//finally write the output data buffer to the HDFS file.
}
```

4. Performance Evaluation

In this section we present the experiments conducted for large size images of remote sensing data having different dimensions (scans, pixels) and sizes varying approximately from 288 Megabytes to 9.1 Gigabytes. First we discuss the read and write performance, storage overheads of the conventional system, both with and without overlapping scan lines, followed by performance comparison of histogram and sobel edge detection filter operations. We conduct the experiments both on conventional APIs and XHAMI libraries, and discuss how XHAMI simplifies the programming complexity and also increases the performance when applied to a large scale image over Hadoop framework.

Table 8.System configuration

Type	Processor type	hostname	RAM (GB)	Disk (GB)
Name node	Intel Xeon 64 bit , 4 vCpus, 2.2 GHz	namenode	4	100
Job tracker	-do-	jobtracker	2	80
Data node 1	-do-	datanode1	2	140
Data node 2	Intel Xeon 64 bit , 4 vCpus, 2.2 GHz	datanode2	2	140
Data node 3	Intel Xeon 64 bit , 2 vCpus, 2.2 GHz	datanode3	2	140
Data node 4	-do-	datanode4	2	100

For the experimental study, we have used virtualized environment running on Xen hypervisor with a pool of four servers of Intel Xeon 64 bit architecture, with 2TB internal storage. Hadoop version 2.7 is configured in the fully distributed mode, running on the server pool of four virtual machines with 64 bit 'Cent OS', the nodes configuration is shown in Table 8.

4.1 Storage overheads

Sample data sets for experiments are described in Table 9, the columns in the table, *Image size* represents the original image size in bytes in regular file system, and the *resulted image size* indicates the size in bytes in HDFS with overlapping of 5 scan lines. A sample image with overlap of 5 scan lines shown in red color is depicted in Figure 4. The results show a maximum of 0.25% increase in the image size, which is negligible.

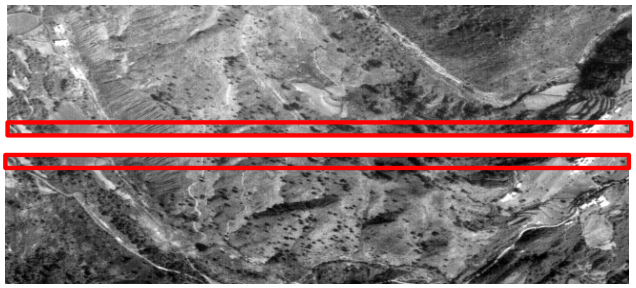


Figure 4.Image blocks with overlap

Table 9. Sample data sets used and the resultant image size

S.No	Image size (in bytes)	Scan line length	Total Scan lines	Resulted Image size (in bytes)
1	288000000	12000	12000	288480000
2	470400000	12000	19600	471240000
3	839976000	12000	34999	841416000
4	1324661556	17103	38726	1327911126
5	3355344000	12000	139806	3361224000
6	9194543112	6026	762906	9202738472

Table 10. Read/write performance overheads

S.No	Image size (MB)	Write (Sec)		Read (Sec)	
		Default Hadoop	XHAMI	Default Hadoop	XHAMI
1	275	5.865	5.958	10.86	10.92
2	449	14.301	14.365	19.32	19.45
3	802	30.417	30.502	40.2	40.28
4	1324	44.406	77.153	50.28	50.95
5	3355	81.353	88.867	90.3	90.6
6	6768	520.172	693.268	550.14	551.6

4.2 Read / write overheads

Performance of write and read function in default Hadoop and XHAMI with overlap of 5 scan lines is shown in Figure 5 and Figure 6 respectively. The results indicate that there is a negligible overhead for both the I/O operations, as the numbers of scan lines to be skipped are very little, and the position of those lines is known prior, hence there is no much performance overhead is

observed. The results indicate that, write function with overlap XHAMI has little performance overheads compared with default Hadoop as shown in Table 10.

The write overheads for data sets in serial nos. 1 , 2, 3 and 5 is less than 5% , and for other data sets it is 33%. Read performance for all the data sets is less than 0.2% which is very negligible.

For data sets 4 and 6 the write overhead is significant, the reason for it is the dimensions of the image is such that , the number of scans is far less than the number of pixels, hence, partitioning of the blocks horizontally with overlapped scan lines is not a optimal while, portioning the blocks in vertical directions is preferred. Hence, we can conclude that the partitioning of the blocks may be chosen based on the dimensions of the image.

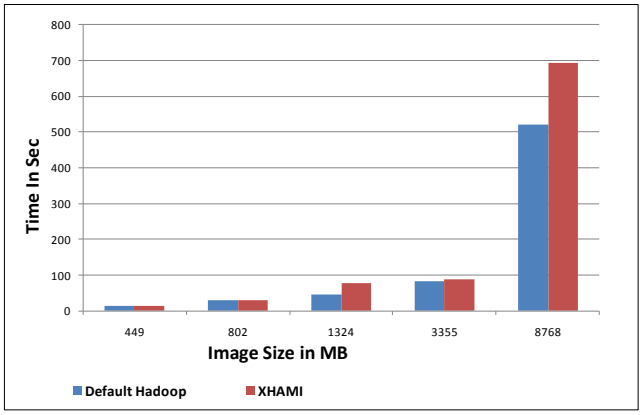


Figure 5. Image write performance

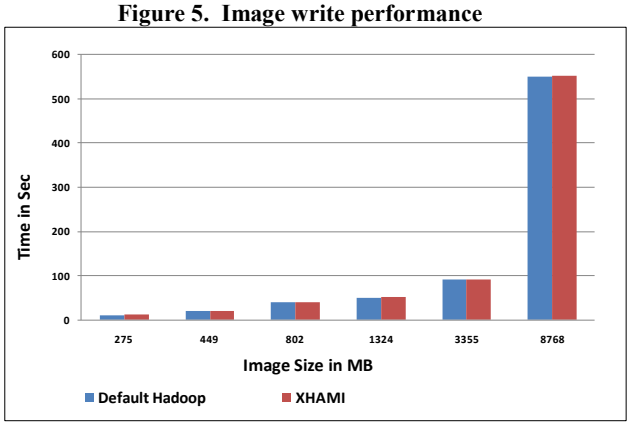


Figure 6. Image read performance

4.3 Performance Comparisons of Map/Reduce

We discuss the performance comparison of default Hadoop and XHAMI for histogram and sobel filter on the data sets mentioned in Table 9.

(a) Histogram operation

Histogram operation counts the frequency of the pixel intensity in the entire image, which is similar to counting the words in the file. However, due to XHAMI data organization the overlapped pixels need to be counted only once which may incur additional overheads.

The performance results of histogram operation of default Hadoop and XHAMI system is shown in Figure 7. The results show that, there is no significant difference in the execution timings, which is less than 0.8%.

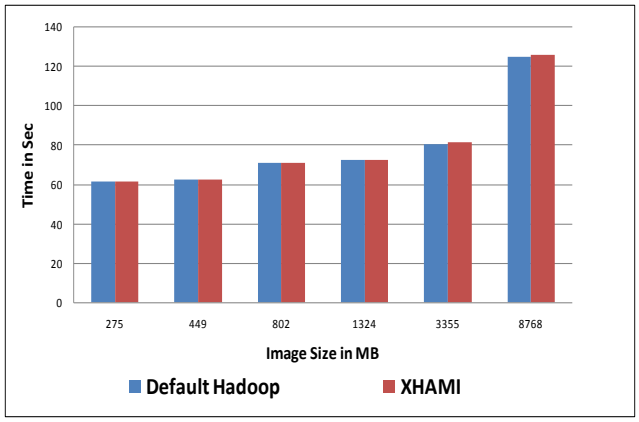


Figure 7. Histogram performance

(b) Fixed mask convolution operation

Convolution methods are most common operation done in image processing. Sobel operator is one of the commonly used method for detecting edges in the image. It involves multiplication of 3X3 mask around each pixel. It is to be noted for the reasons mentioned in the section 1, default Hadoop implementation cannot produce required result, however, to overcome this limitation, it is necessary to introduce additional I/O operations from the adjacent blocks.

The performance of the sobel edge detection shown in Figure 8, illustrates that execution time of XHAMI. It is to be noted that XHAMI implementation nearly takes the half of the time compared to default Hadoop. This is because due to the overlap pixels are organized within the blocks. Apart from this, the default Hadoop system requires more programming complexity like reading the image, writing it to the blocks, and reading the overlapping neighborhood pixels etc, these processes are offered as high level APIs by XHAMI system, which not only simplifies the programming complexity but also allows the development of image processing applications rapidly on Hadoop framework.

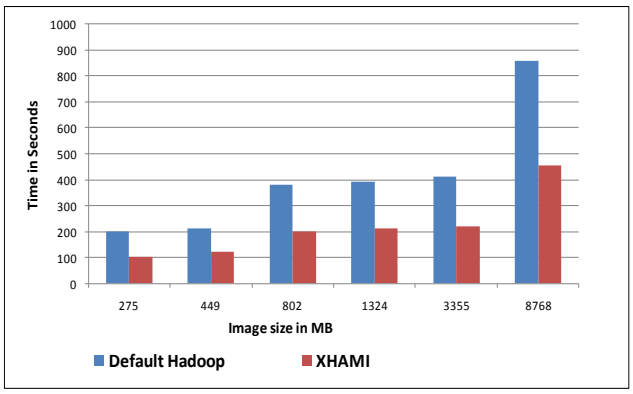


Figure 8. Sobel filter performance

5. Conclusions and Future Work

Image processing applications deal with processing of pixels in parallel, for which Hadoop and MapReduce can be effectively used to obtain higher throughputs. However many of the algorithms in Image Processing and other scientific computing, require use of neighborhood data, for which the existing methods of data organization and processing are not suitable. We presented an extended HDFS and MapReduce interface, called XHAMI, for image processing applications. XHAMI offers extended library of HDFS and MapReduce to process the single large scale images with high level of abstraction over writing and reading the images. APIs are offered for all the basic forms Read/Write and Query of images. Several experiments are conducted on sample of six data sets with a single large size image varying from approximately 288 MB to 9.1 GB.

Several experiments are conducted for reading and writing the images with and without overlap using XHAMI. The experimental results are compared with the conventional Hadoop system, the experimental results show that, though the proposed methodology incurs marginal read and write overheads, due to overlapping of data, the performance has scaled linearly and also programming complexity is reduced significantly.

Currently, the system is implemented with the fixed length record; in future it is proposed to use the customized split function for processing, which would allow spawning more map functions for processing. However, challenges involved in organizing the sequence of executed map functions for aggregations need to be addressed. We plan to implement the bi-directional split also in the proposed system, which would be the requirement for large scale canvas images. The proposed MapReduce APIs could be extended for many more Image processing and Computer vision modules. It is also proposed to extend the same to multiple image formats in the native format itself.

Currently, image files are transferred one at a time from the local storage to Hadoop cluster. In future, Data aware scheduling discussed in our earlier work [28] will be integrated for the large scale data transfers from the replicated remote storage repositories and performing group scheduling on the Hadoop cluster.

Acknowledgements

We express our thanks to Mr. Nooka Ravi of ADRIN for discussion on GDAL library, and Mr. Rama Krishna Reddy V. for discussions on image filters based on spatial and frequency domains. We thank Smt. GeetaVaradan, Director of ADRIN for her support and encouragement in pursuing the research.

References

[1] IDC, The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest growth in the Far East, www.emc.com/leadership/digital-universe/index.htm.

[2] K.Bakshi, Considerations for Big Data: Architecture and Approach Aerospace Conference- Big Sky, MT, 3-10 March 2012.

[3] K. Michael, and K. W. Miller, Big Data: New opportunities and New Challenges, IEEE Computer, 46 (6) (2013): 22-24.

[4] The Apache Hadoop Project, http://hadoop.apache.org

[5] J. Kelly, Big Data: Hadoop, Business Analytics and Beyond, Wikibon White paper, 27th August 2012, http://wikibon.org/wiki/v/Big Data: Hadoop, Business Analytics and Beyond.

[6] S. Ghemawat, H. Gobioff, and S. T. Leung, *The Google File System,* In Proc. 9th ACM symposium on Operating System Principles (SOSP 2003),NY, USA 2003, pp. 29-43.

[7] K. Schvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: Proc. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 2010), Incline Village, Nevada, USA, May 2010.

[8] J. Dean, and S. Ghemat, MapReduce: Simplified Data Processing on Large Cluster, Communications of the ACM, 51(1) (2008): 107-113.

[9] C. Jin, and R. Buyya, *MapReduce Programming Model for .NET-Based Cloud Computing,* Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science, 5704 (2009): 417- 428.

[10] J. Ekanayake, S. Pallickara, and G. Fox, MapReduce for datat intensive analyses, in: Proc. IEEE fourth International Conference on e-Science, Indiana Police, Indiana, USA, Dec 2008, pp. 277-284.

[11] Satellite imaging corporation, http://www/satimagingcorp.com/satellite-sensors

[12] Y. Ma, H. Wu, L. Wang, B. Huang, R. Ranjan, A. Zomaya, and W. Jie, Remote Sensing Big Data computing: Challenges

and opportunities, Future Generation Computer Systems, Volume 51, October 2015, Pages 47–60.

[13] R. C. Gonzalez, and R. E. Woods, Digital Image Processing, 3rd Edition, 2007 (chapters 1, and 3).

[14] X. Cao, S. Wang, Research about Image Mining Technique, Journal of Communications in Computer and Information Sciences, 288(2012): 127-134.

[15] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox, Twister: A Runtime for Iterative MapReduce, in: Proc. 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010),Chicago, Illinois, USA 2010,pp. 810-818.

[16] W. Jiang, V. T. Ravi, G. Agarwal, A Map-Reduce System with an Alternate API for Multi-Core Environments, in: Proc. 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID 2010), Melbourne, May 2010,pp. 84-93.

[17] L. Kennedy, M. Slaney, and K. Weinberger, Reliable Tags using Image Similarity: Mining Specificity and Expertise from Large-Scale Multimedia Databases, in: Proc. 1st workshop on Web-Scale Multimedia Corpus, Beijing, October 2009, pp. 17-24.

[18] L. L. Shi, B. Wu, B. Wang, and X. G. Yang, Map/Reduce in CBIR Applications, in: Proc. International Conference on Computer Science and Network Technology (ICCSNT), Harbin, December 2011, pp. 2465-2468.

[19] C. T. Yang, L. T. Chen, W. L. Chou, and K. C. Wang, Implementation on Cloud Computing, in: Proc. IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS 2011), Beijing, September 2011, pp. 482-485.

[20] H. Kocalkulak, and T. T. Temizel, A Hadoop Solution for Ballistic Image Analysis and Recognition, in: Proc. International Conference on High Performance Computing and Simulation (HPCS 2011), Istanbul, July 2011, pp. 836-842.

[21] M. H. Almeer, Cloud Hadoop MapReduce For Remote Sensing Image Analysis, Journal of Emerging Trends in Computing and Information Sciences, vol. 3, 637-644.

[22] I. Demir, and A. Sayar, Hadoop Optimization for Massive Image Processing: Case Study of Face Detection, International Journal of Computers and Communications and Control, 9(6) (2014), 664-671.

[23] T. White, The Small files problem: 2009, http://www.cloudera.com/blog/2009/02/02/the-small-files-problem.

[24] C. Sweeney, L. Liu, S. Arietta, and J. Lawrence, HIPI: A Hadoop Image Processing Interface for Image-based MapReduce Tasks, undergraduate thesis, University of Virginia, USA.

[25] P. Jakovits, and S. N. Srirama, Large Scale Image Processing Using MapReduce, thesis, Tartu University, 2013.

[26] GDAL, Gdal- geospatial data abstraction library.

[27] Apache HBASE project, http://www.hbase.apache.org

[28] R. Kune, K. P. Kumar, A. Agarwal, C. R. Rao, and R. Buyya, Genetic Algorithm based Data-aware GroupScheduling for Big Data Clouds, in: Proc. International Symposium on Big Data Computing (BDC 2014), London, December 2014, pp. 96-104.